

Advanced Java Week 3 Day 01

Thread & Multithreading In Java And Collections

Agenda:

1. [Threads In Java](#)
2. [Multithreading - Thread Based Multitasking](#)
3. [Multithreading fundamentals](#)
4. [Creating Thread](#)
5. [Implementing the Runnable interface](#)
6. [Thread States:](#)
7. [Creating Multiple Threads](#)
8. [Determining When a Thread Ends](#)
9. [Thread Priorities](#)
10. [Synchronization](#)
11. [Synchronized Keyword](#)
12. [Synchronized void display \(String msg\)](#)
13. [Using Synchronized block](#)
14. [Inter Thread Communication \(wait, notify, notifyAll\)](#)
15. [Deadlock](#)
16. [Suspending, Resuming and Stopping Threads](#)

Threads In Java

Threading is a very useful feature in any Programming language and Java provides a very effective and efficient Threading mechanism.

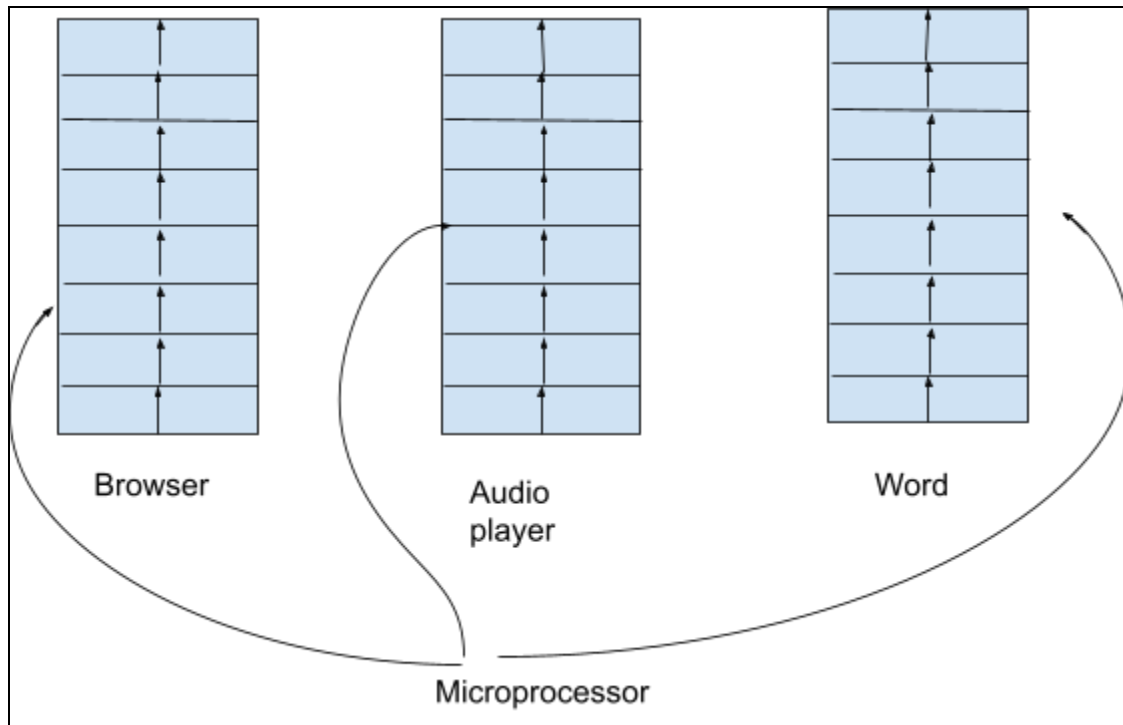
To Understand what a thread is, let us understand the feature of Multitasking.

Multitasking is a feature by which we can execute multiple tasks simultaneously, this is achieved through the following features.

- 1) Subtasking
- 2) Time slicing
- 3) Time sharing

Subtasking is a feature by which the task is broken down into multiple sub tasks, for example when we are doing multiple tasks like Browsing, using a browser, Listening to a music using an audio player and in parallel editing word document, all these applications are running in parallel,

This is achieved by the above mentioned features.



Each of the tasks is broken down into subtasks as illustrated in the diagram above and the Time is also sliced, in the sense, first fraction of a second, say a Billionth of a second, Processor pays its attention to the First task i.e Browser and the next fraction of a second it pays its attention to Audio player and the next fraction of a second , it pays its attention to the Word and this process continues repetitively until one of the app (task) is terminated or closed explicitly in a round robin fashion and the net result is the user feels as if all the tasks are running in parallel, but there has been a path of execution inside every task , this overall **path of execution** is nothing but termed as **Thread**

Hence a Thread is nothing but a **path of execution** of a **Process**, a Process is nothing but an Application under execution.

We can say a small amount of time is being shared among multiple processes, thus MultiTasking is achieved through SubTasking (Breaking the tasks into multiple subtasks) , Time Slicing (Time is broken into small unit) & Time Sharing (smallest unit of time is shared across multiple tasks)

Multitasking can take place in 2 ways,

- 1) Process Based Multitasking
- 2) Thread based Multitasking

Above illustration is an example of **Process based multitasking**, i.e each of the above Processes is executed in its own individual memory space.

Multithreading - Thread Based Multitasking

But the same phenomenon which is described above can take place within a **single process**, that makes it Thread Based Multitasking,

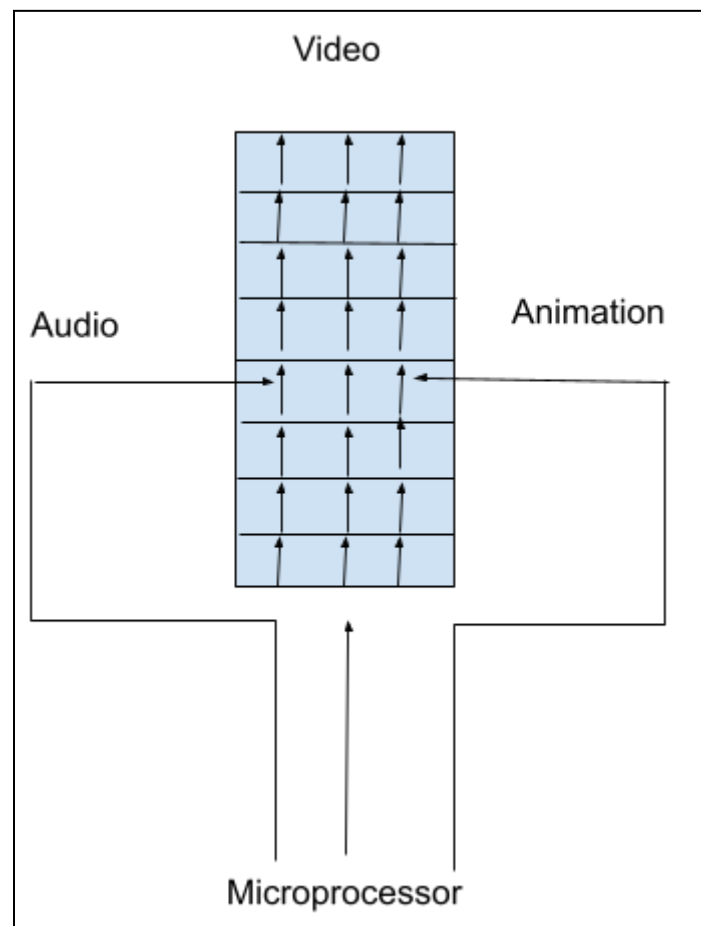
In a Thread Based Multitasking , there are multiple Threads within a single Process, i.e multiple paths of executions within a single Process.

Say for example in a Multimedia based application audio file , a video file and an animation based function is executed within a single process, but all the three functionalities are to be executed in parallel, it is not that the audio file executes and terminates then the video app starts.

Therefore the processor pays its attention to all the three functionalities in a round robin fashion as illustrated above but within a single Process, because three functionalities run as separate threads.

Thus it becomes a Multithreaded application.

Every application has a main thread/primary thread along with which we can assign multiple threads within the process in order to execute multiple tasks simultaneously as well as use least resource like Memory in an efficient manner.



Multithread based applications provide the following advantages.

- 1) Executes Multiple tasks simultaneously
- 2) Improves performance
- 3) Uses efficient memory.

Threads in Java can be implemented by one of the following 2 methods

- 1) Using Thread class
- 2) Using Runnable Interface

Multithreading fundamentals

In Java, multithreading is the process of running several threads at the same time.

A multithreaded program is made up of two or more components that can execute at the same time. Each component of such a program is referred to as a thread, and each thread defines a separate route of execution. Thus, multithreading is a subset of multitasking.

there are two distinct types of multitasking: process-based and thread-based as discussed above

A process is, in essence, a running code. As a result, process-based multitasking is a feature that allows your computer to execute two or more programs concurrently. Process-based multitasking, for example, allows you to run the Java compiler while also using a text editor or browsing the web.

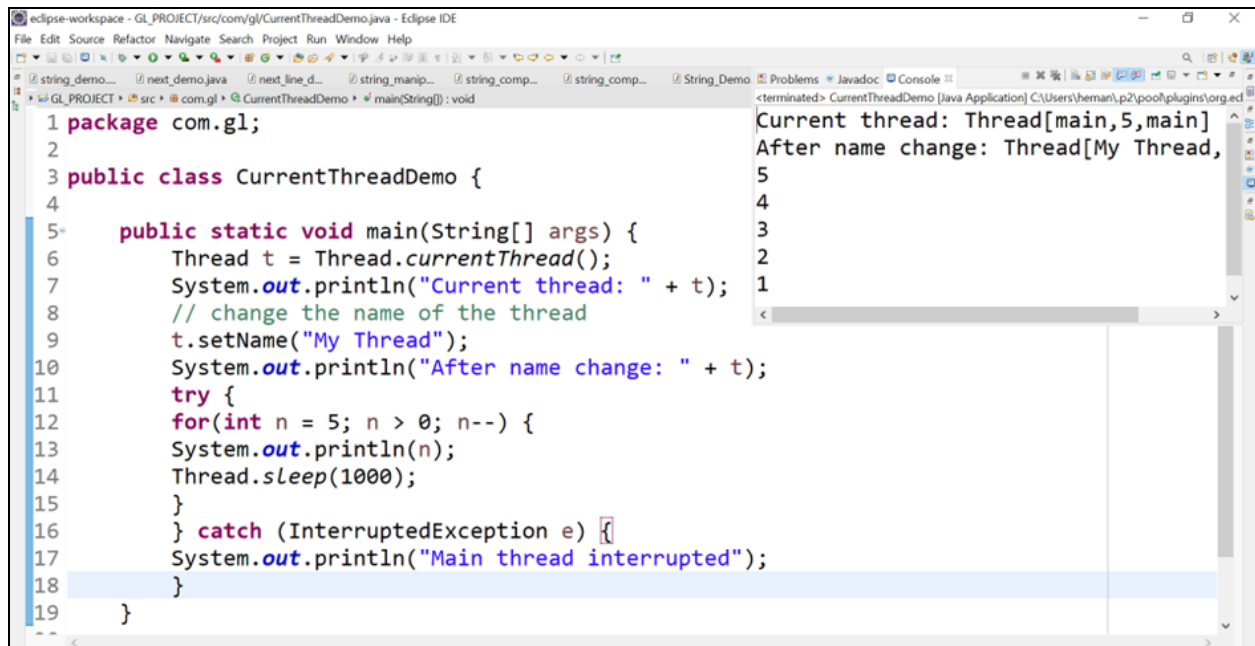
In a thread-based multitasking system, the thread is the smallest unit of dispatchable code. This means that a single software may accomplish two or more jobs at the same time. For example, a text editor can format text while printing, as long as the two activities are executed by distinct threads.

The Main Thread

When a Java program starts, one thread runs immediately. This is sometimes referred to as your program's main thread since it is the one that is run when your program starts. The main thread is essential for two reasons:

1. It is the thread from which additional "child" threads are born.
2. It is frequently the last thread to complete execution since it conducts different shutdown operations.

Example:

The screenshot shows the Eclipse IDE interface. The main editor window displays a Java file named 'CurrentThreadDemo.java' with the following code:

```
1 package com.gl;  
2  
3 public class CurrentThreadDemo {  
4  
5     public static void main(String[] args) {  
6         Thread t = Thread.currentThread();  
7         System.out.println("Current thread: " + t);  
8         // change the name of the thread  
9         t.setName("My Thread");  
10        System.out.println("After name change: " + t);  
11        try {  
12            for(int n = 5; n > 0; n--) {  
13                System.out.println(n);  
14                Thread.sleep(1000);  
15            }  
16        } catch (InterruptedException e) {}  
17        System.out.println("Main thread interrupted");  
18    }  
19 }
```

The right-hand side of the IDE shows the 'Console' view with the following output:

```
<terminated> CurrentThreadDemo [Java Application] C:\Users\heman\p2\pool\plugins\org.ed  
Current thread: Thread[main,5,main]  
After name change: Thread[My Thread,  
5  
4  
3  
2  
1
```

In the above program,

The main thread is created automatically when the program is executed, and it may be controlled using a Thread object.

To do so, you must first gain a reference to it by invoking the `currentThread()` function, which is a public static member of Thread.

Syntax:

`static Thread currentThread()`

In this program, executing `currentThread()` returns a reference to the current thread (the main thread in this example), which is then stored in the local variable `t`.

The program then shows thread information. The program then runs `setName()` to update the thread's internal name. The thread's information is then re-displayed.

A loop then counts down from five, stopping one second between each line. The `sleep()` function is responsible for the pause.

The argument to `sleep()` specifies the delay period in milliseconds.

Try/catch block is used to handle the `InterruptedException` which might be thrown by sleep method

Creating Thread

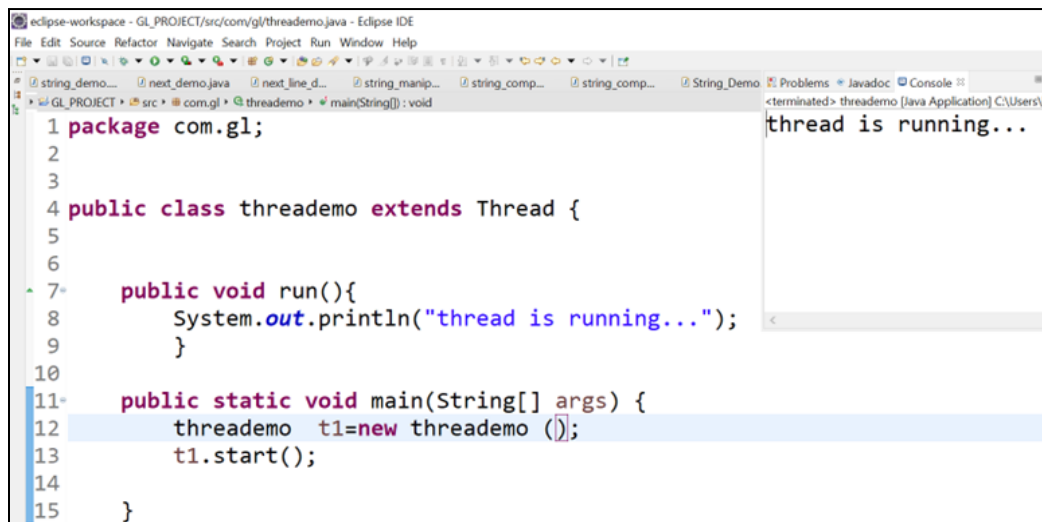
There are two ways to create threads in java

1. By extending the Thread class, itself
2. implementing the Runnable interface.

Extend the Thread class, itself

In this method, a new class that extends the Thread class is created and then create an instance of that class. The extending class must override the run() method, which serves as the new thread's entry point. It must also call start() to start the new thread's execution.

Example:



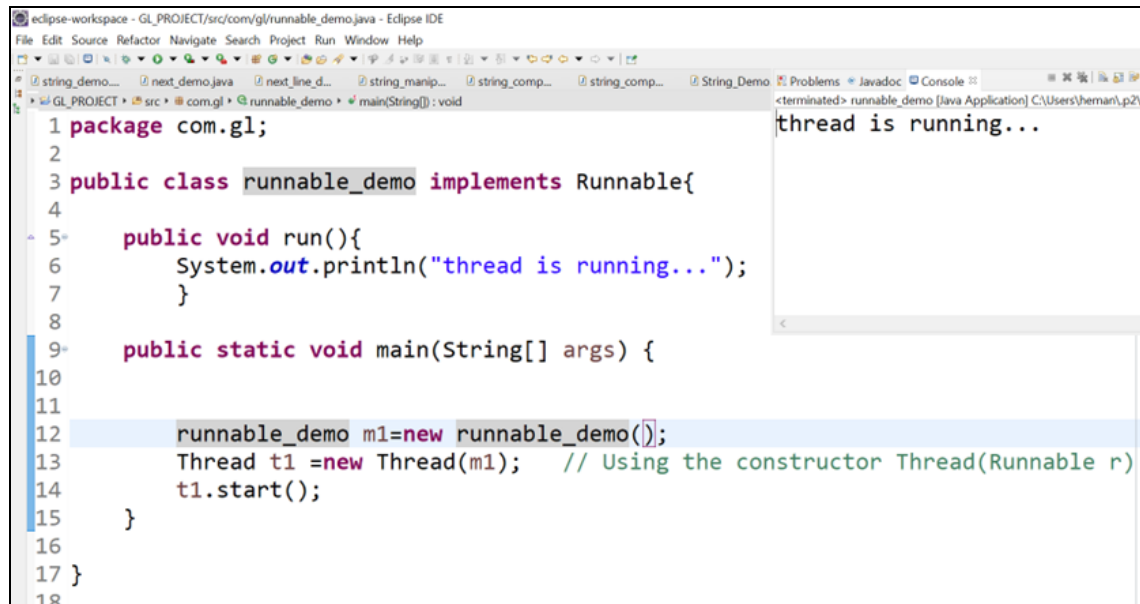
```
1 package com.gl;
2
3
4 public class threaddemo extends Thread {
5
6
7     public void run(){
8         System.out.println("thread is running...");
9     }
10
11     public static void main(String[] args) {
12         threaddemo t1=new threaddemo ();
13         t1.start();
14     }
15 }
```

The screenshot shows the Eclipse IDE with a Java file named 'threaddemo.java'. The code defines a class 'threaddemo' that extends 'Thread'. It overrides the 'run()' method to print 'thread is running...' and includes a 'main' method that creates an instance of 'threaddemo' and calls 'start()'. The console on the right shows the output 'thread is running...'.

Implementing the Runnable interface

The most straightforward way to create a thread is to write a class that implements the Runnable interface. A unit of executable code is abstracted by runnable. A thread can be created on any object that implements Runnable. A class that implements Runnable must only implement a single method called run ()

The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:



```
1 package com.gl;
2
3 public class runnable_demo implements Runnable{
4
5     public void run(){
6         System.out.println("thread is running...");
7     }
8
9     public static void main(String[] args) {
10
11
12         runnable_demo m1=new runnable_demo();
13         Thread t1 =new Thread(m1);    // Using the constructor Thread(Runnable r)
14         t1.start();
15     }
16
17 }
18
```

thread is running...

Thread States:

- 1) UnStarted
- 2) Started / Runnable
- 3) Not Runnable
 - Running
- 4) Terminated

When an Object of Thread is created, thread is said to be in “Unstarted” state

Thread t1 = new Thread

The above Thread is in **UnStarted** State

When t1.start() is called , it automatically invokes run() method, then the thread enters **Runnable/Started** state

But not necessarily the Thread is in Running state, though run method is to be invoked for the respective thread, only if the Thread gets its priority, then the Thread goes to **Running** state, wherein its **run** method would be functional.

Once the thread is in running state, it might be suspended for some reason or made to wait to get the lock or might be in waiting state because of its low priority, in all such situations, the thread is said to be in **“Not Runnable”** state

Once the Thread has completed its task or if explicitly the thread is aborted, it is said to be in **“Terminated”** state

Creating Multiple Threads

The following program, for example, generates three child threads:

```
eclipse-workspace - GL_PROJECT/src/com/gl/mutli_threads.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
GL_PROJECT > src > com.gl > NewThread > t: Thread
1 package com.gl;
2 //Create multiple threads.
3 class NewThread implements Runnable {
4     String name; // name of thread
5     Thread t;
6     NewThread(String threadname)
7     {
8         name = threadname;
9         t = new Thread(this, name);
10        System.out.println("New thread: " + t);
11        t.start(); // Start the thread
12    }
13    //This is the entry point for thread.
14    public void run()
15    {
16        try {
17            for(int i = 5; i > 0; i--)
18            {
19                System.out.println(name + ": " + i);
20                Thread.sleep(1000);
21            }
22        }
23        catch (InterruptedException e)
24        {
25            System.out.println(name + "Interrupted");
26        }
27        System.out.println(name + " exiting.");
28    }
29 }
```

```
30
31 public class mutli_threads
32 {
33     public static void main(String args[])
34     {
35         new NewThread("One"); // start threads
36         new NewThread("Two");
37         new NewThread("Three");
38         try {
39             // wait for other threads to end
40             Thread.sleep(10000);
41         } catch (InterruptedException e) {
42             System.out.println("Main thread Interrupted");
43         }
44         System.out.println("Main thread exiting.");
45     }
46 }
47
```


Output:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
Two: 4
One: 4
Three: 4
Two: 3
One: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
One exiting.
Three exiting.
Main thread exiting.
```

The CPU is shared by all three child threads in the above program. The main method is put to sleep for 10 seconds by calling `sleep(10000)`.

Determining When a Thread Ends

There are two ways to tell if a thread has finished. First, you can check the thread's status by calling `isAlive()`. Thread defines this method, and its general form is shown below:

```
final boolean isAlive()
```

If the thread on which the `isAlive()` method is called is still running, it returns `true`. Otherwise, it returns `false`.

While `isAlive()` is occasionally useful, the more common method for waiting for a thread to finish is `join()`, as shown here:

```
final void join() throws InterruptedException
```

This method waits until the thread on which it is called terminates

Example :

```
eclipse-workspace - GL_PROJECT/src/com/gl/thread_demo.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

GL_PROJECT > src > com.gl > NewThread1 > run(): void

1 package com.gl;
2
3 //Using join() to wait for threads to finish.
4 class NewThread1 implements Runnable {
5     String name; // name of thread
6     Thread t;
7     NewThread1(String threadname)
8     {
9         name = threadname;
10        t = new Thread(this, name);
11        System.out.println("New thread: " + t);
12        t.start(); // Start the thread
13    }
14    //This is the entry point for thread.
15    public void run()
16    {
17        try {
18            for(int i = 5; i > 0; i--) {
19                System.out.println(name + ": " + i);
20                Thread.sleep(1000);
21            }
22        }
23        catch (InterruptedException e) {
24            System.out.println(name + " interrupted.");
25        }
26        System.out.println(name + " exiting.");
27    }
28 }
29
```

```
37
38     System.out.println("Thread One is alive: "
39         + ob1.t.isAlive());
40     System.out.println("Thread Two is alive: "
41         + ob2.t.isAlive());
42     System.out.println("Thread Three is alive: "
43         + ob3.t.isAlive());
44     // wait for threads to finish
45     try {
46         System.out.println("Waiting for threads to finish.");
47         ob1.t.join();
48         ob2.t.join();
49         ob3.t.join();
50     } catch (InterruptedException e) {
51         System.out.println("Main thread Interrupted");
52     }
53     System.out.println("Thread One is alive: "
54         + ob1.t.isAlive());
55     System.out.println("Thread Two is alive: "
56         + ob2.t.isAlive());
57     System.out.println("Thread Three is alive: "
58         + ob3.t.isAlive());
59     System.out.println("Main thread exiting.");
60 }
61
62 }
```

Output:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Thread One is alive: true
Thread Two is alive: true
Two: 5
Three: 5
Thread Three is alive: true
Waiting for threads to finish.
Two: 4
Three: 4
One: 4
Two: 3
Three: 3
One: 3
Three: 2
Two: 2
One: 2
Three: 1
Two: 1
One: 1
Three exiting.
Two exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

In the above program, after the calls to `join()` return, the threads have stopped executing.

Thread Priorities

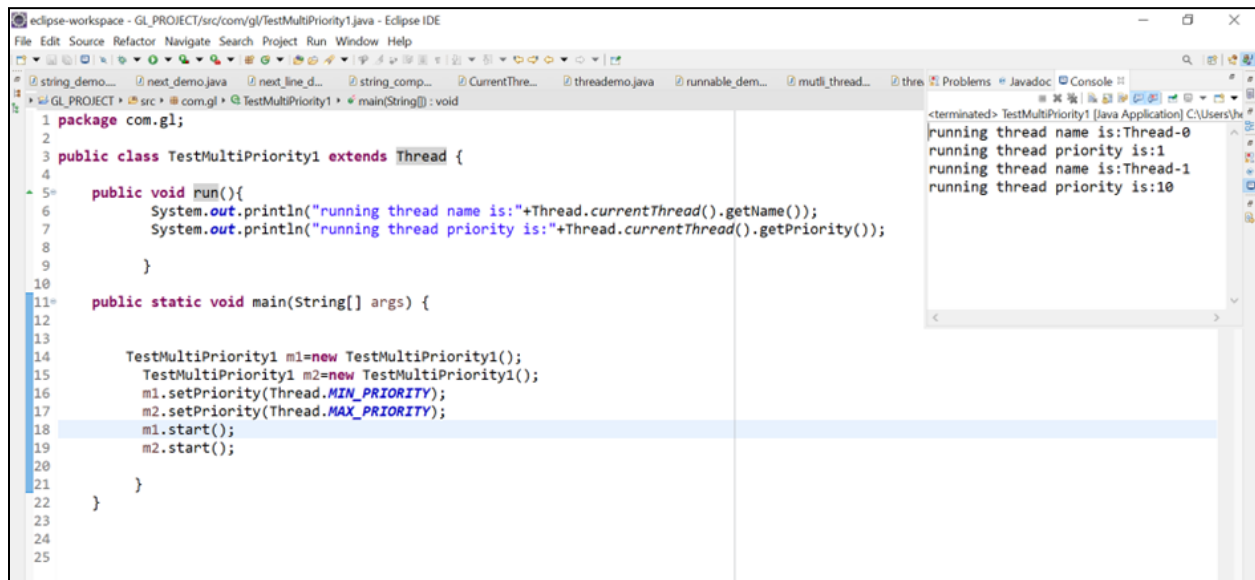
There is a priority for each thread. Priorities are assigned a number between one and ten. Thread scheduling, in most cases, schedules threads according to their priority (known as preemptive scheduling). However, it is not guaranteed because the scheduling option chosen by the JVM is dependent on the JVM specification.

3 constants defined in Thread class:

```
public static int MIN_PRIORITY
public static int NORM_PRIORITY
```

```
public static int MAX_PRIORITY
```

A thread's default priority is 5 (NORM_PRIORITY). The MIN_PRIORITY value is 1 and the MAX_PRIORITY value is 10.



The screenshot shows the Eclipse IDE with a Java project named 'GL_PROJECT'. The main editor displays the source code for 'TestMultiPriority1.java'. The code defines a class that extends 'Thread' and implements a 'run()' method that prints the current thread's name and priority. The 'main()' method creates two instances of 'TestMultiPriority1', sets their priorities to 'MIN_PRIORITY' and 'MAX_PRIORITY', and starts them. The 'Console' window on the right shows the output of the program, which is terminated. The output shows four lines: 'running thread name is:Thread-0', 'running thread priority is:1', 'running thread name is:Thread-1', and 'running thread priority is:10'.

```
1 package com.gl;
2
3 public class TestMultiPriority1 extends Thread {
4
5     public void run(){
6         System.out.println("running thread name is:"+Thread.currentThread().getName());
7         System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
8     }
9
10
11     public static void main(String[] args) {
12
13         TestMultiPriority1 m1=new TestMultiPriority1();
14         TestMultiPriority1 m2=new TestMultiPriority1();
15         m1.setPriority(Thread.MIN_PRIORITY);
16         m2.setPriority(Thread.MAX_PRIORITY);
17         m1.start();
18         m2.start();
19     }
20
21 }
22
23
24
25
```

```
<terminated> TestMultiPriority1 [Java Application] C:\Users\h...
running thread name is:Thread-0
running thread priority is:1
running thread name is:Thread-1
running thread priority is:10
```

Output:

```
running thread name is:Thread-0
running thread priority is:1
running thread name is:Thread-1
running thread priority is:10
```

Synchronization

The process of handling multiple thread requests for resource access is known as synchronization. Synchronization's main goal is to prevent thread interference. We must ensure that a shared resource is only used by one thread at a time when multiple threads attempt to access it. This is accomplished through the process of synchronization. The synchronization keyword in Java generates a critical section, which is a code block, paused until the first thread finishes. The monitor is said to be waiting for these additional threads. A thread that owns a monitor can choose to re-enter that monitor if it so desires.

General Syntax:

```
synchronized (object)
{
    //statement to be synchronized
}
```

}

Synchronization can be achieved in two ways

Using **synchronization** method

Using **synchronized** statement

Why we need synchronization

The results will be distorted if we do not use synchronization and allow two or more threads to access a shared resource at the same time.

Assume T1 and T2 are two threads. T1 starts running and saves some values to a file called temporary.txt, which will be used to calculate a result when T1 returns. T2 starts in the meantime, and before T1 returns, T2 changes the values saved in the temporary.txt file by T1 (temporary.txt is the shared resource). T1 will now, without a doubt, produce an incorrect result.

To avoid such problems, synchronization was implemented. When using synchronization, the temporary.txt file is locked (LOCK mode) once T1 starts using it. No other thread can access or modify T1 returns

There are two ways to achieve synchronization

Example with no Synchronization

In this example, we are not using synchronization and are instead spawning multiple threads that access the display method and generate random output.

```
eclipse-workspace - GL_PROJECT/src/com/gl/no_sync_demo.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
GL_PROJECT src com.gl First display(String) : void
1 package com.gl;
2
3
4 class First
5 {
6     public void display(String msg)
7     {
8         System.out.print ("["+msg);
9         try
10        {
11            Thread.sleep(1000);
12        }
13        catch (InterruptedException e)
14        {
15            e.printStackTrace();
16        }
17        System.out.println ("]");
18    }
19 }
20
21 class Second extends Thread
22 {
23     String msg;
24     First fobj;
25     Second (First fp,String str)
26     {
27         fobj = fp;
28         msg = str;
29         start();
30     }
31     public void run()
32     {
33         fobj.display(msg);
34     }
35 }
36
37 public class no_sync_demo {
38
39     public static void main(String[] args) {
40         First fnew = new First();
41         Second ss = new Second(fnew, "welcome");
42         Second ss1= new Second(fnew,"new");
43         Second ss2 = new Second(fnew, "programmer");
44     }
45 }
46 }
47
```

Output:

[welcome[new[programmer]]]

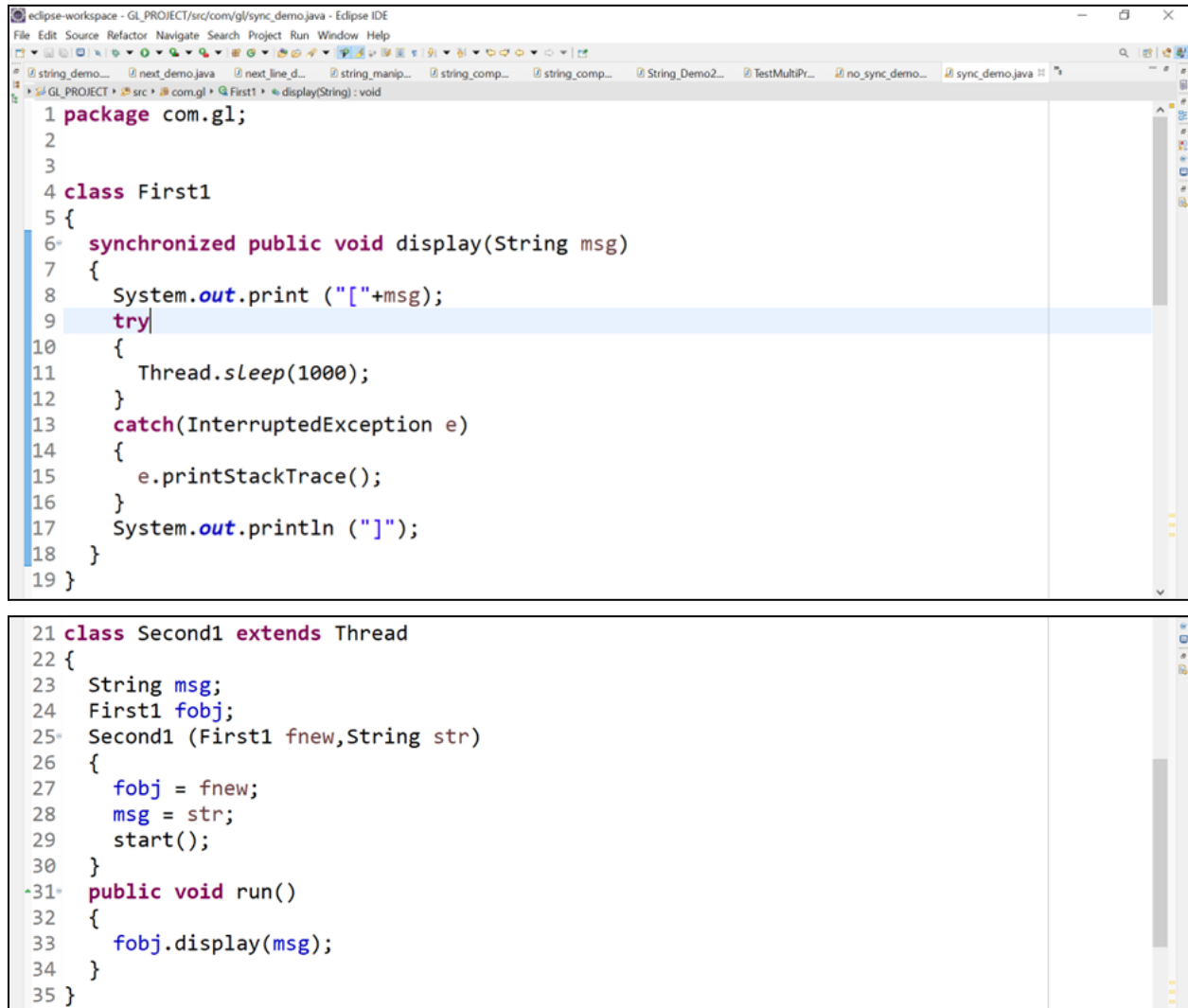
The object fnew of class First is shared by all three running threads (ss, ss1 and ss2) in the above program to call the shared method (void display). As a result, the result is a non synchronized situation known as a race condition.

Synchronized Keyword

To synchronize the above program, we must restrict access to the shared display() method to only one thread at a time. This is accomplished by using the keyword synchronized with the display() method.

Synchronized void display (String msg)

Example : implementation of synchronized method

The image shows a screenshot of the Eclipse IDE interface. The top part of the image displays the source code for a Java class named 'First1' in the package 'com.gl'. The code defines a 'display' method that is synchronized and public. Inside the method, it prints a message, sleeps for 1000 milliseconds, and catches any InterruptedException. The bottom part of the image shows the source code for a 'Second1' class that extends 'Thread'. It has a 'run' method that calls the 'display' method of a 'First1' object. The code is as follows:

```
1 package com.gl;
2
3
4 class First1
5 {
6     synchronized public void display(String msg)
7     {
8         System.out.print ("["+msg);
9         try
10        {
11            Thread.sleep(1000);
12        }
13        catch(InterruptedException e)
14        {
15            e.printStackTrace();
16        }
17        System.out.println ("]");
18    }
19 }
21 class Second1 extends Thread
22 {
23     String msg;
24     First1 fobj;
25     Second1 (First1 fnew,String str)
26     {
27         fobj = fnew;
28         msg = str;
29         start();
30     }
31     public void run()
32     {
33         fobj.display(msg);
34     }
35 }
```

```

36
37 public class sync_demo {
38
39     public static void main(String[] args) {
40         First1 fnew = new First1();
41         Second1 ss = new Second1(fnew, "welcome");
42         Second1 ss1= new Second1(fnew,"new");
43         Second1 ss2 = new Second1(fnew, "programmer");
44     }
45 }
46
47 }

```

Output:

```

[welcome]
[programmer]
[new]

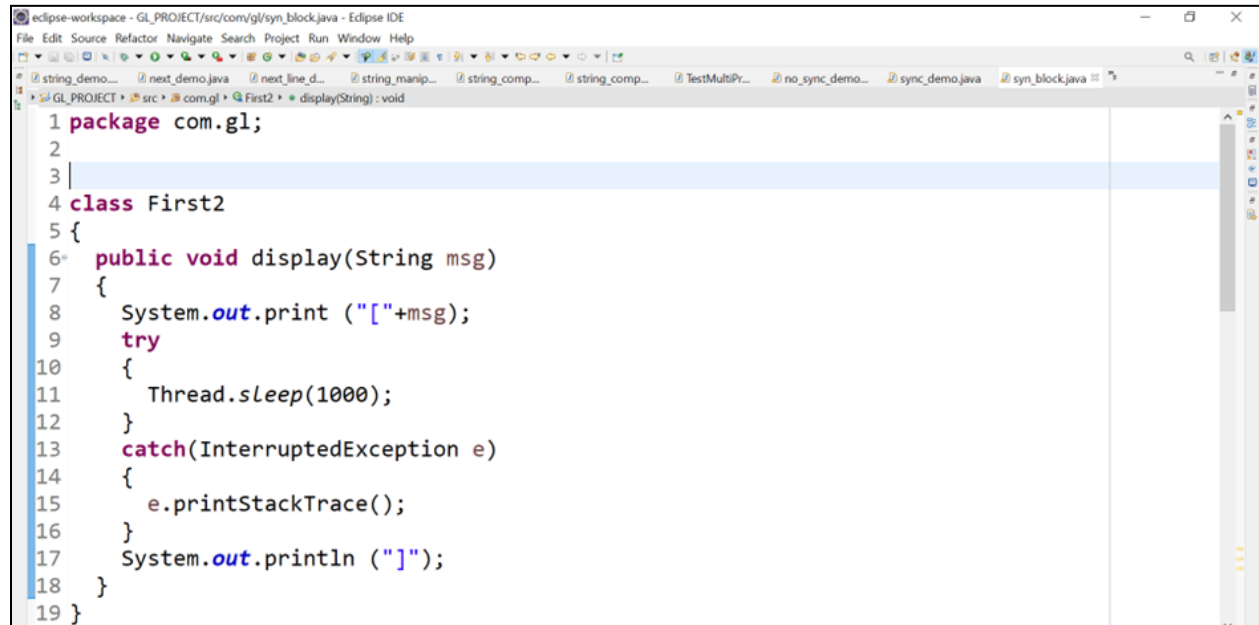
```

Using Synchronized block

A synchronized block can be used to synchronize access to a class object or only a portion of a method. It is capable of synchronizing any part of the object and method.

Example:

In this example, we'll use a synchronized block to limit access to the display method to only one thread at a time.



```

1 package com.gl;
2
3
4 class First2
5 {
6     public void display(String msg)
7     {
8         System.out.print ("["+msg);
9         try
10        {
11            Thread.sleep(1000);
12        }
13        catch (InterruptedException e)
14        {
15            e.printStackTrace();
16        }
17        System.out.println ("]");
18    }
19 }

```



```

20
21 class Second2 extends Thread
22 {
23     String msg;
24     First2 fobj;
25     Second2 (First2 fnew,String str)
26     {
27         fobj = fnew;
28         msg = str;
29         start();
30     }
31     public void run()
32     {
33         synchronized(fobj)    //Synchronized block
34         {
35             fobj.display(msg);
36         }
37     }
38 }

```

```

39
40 public class syn_block {
41
42     public static void main(String[] args) {
43         First2 fnew = new First2();
44         Second2 ss = new Second2(fnew, "welcome");
45         Second2 ss1= new Second2 (fnew,"new");
46         Second2 ss2 = new Second2(fnew, "programmer");
47
48     }
49
50 }

```

Output:

```

[welcome]
[programmer]
[new]

```

Inter Thread Communication (wait, notify, notifyAll)

- The ability of synchronized threads to communicate with one another is referred to as inter-thread communication, also known as cooperation.
- Cooperation (inter-thread communication) is a mechanism that allows one thread to enter (or lock) in the same critical section to be executed while another thread is paused in its critical section. It is implemented using the following methods of the Object class:wait()
- notify()
- notifyAll()
- wait()

The `wait()` method causes the current thread to release the lock and wait until another thread invokes the `notify()` or `notifyAll()` methods for this object, or until a timer expires.

Because this object's monitor must be owned by the current thread, it must be called from the synchronized method only; otherwise, an exception will be thrown.

notify()

The `notify()` method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The decision is arbitrary and is made at the discretion of the implementation.

notifyAll()

Wakes up all threads that are waiting on this object's monitor.

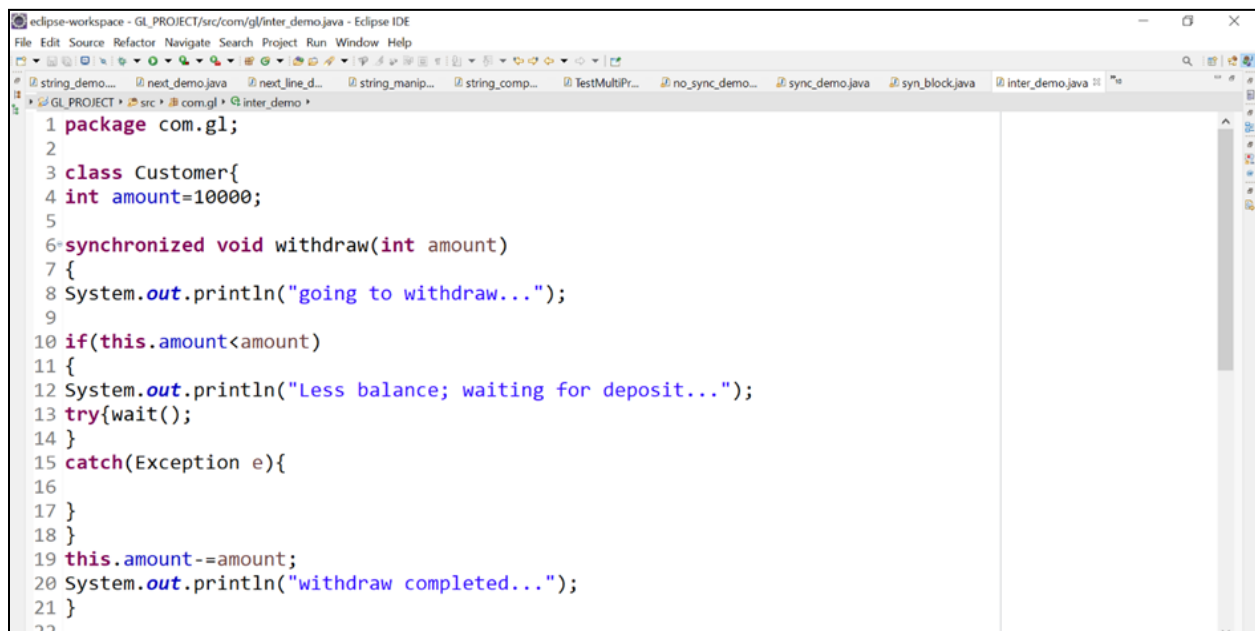
These methods are declared within `Object`, as shown here:

`final void wait()` throws `InterruptedException`

`final void notify()`

`final void notifyAll()`

Example:



```
1 package com.gl;
2
3 class Customer{
4     int amount=10000;
5
6     synchronized void withdraw(int amount)
7     {
8         System.out.println("going to withdraw...");
9
10        if(this.amount<amount)
11        {
12            System.out.println("Less balance; waiting for deposit...");
13            try{wait();}
14        }
15        catch(Exception e){}
16
17    }
18 }
19 this.amount-=amount;
20 System.out.println("withdraw completed...");
21 }
22
```

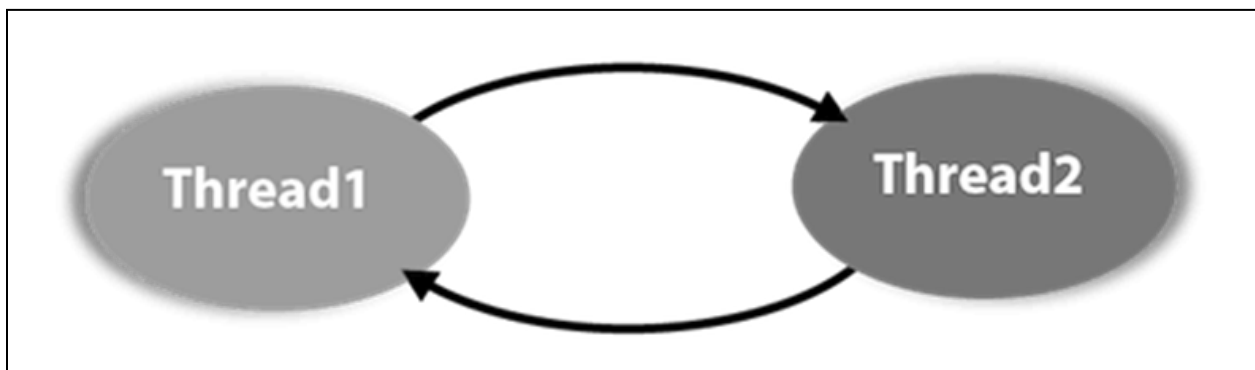
```

23 synchronized void deposit(int amount)
24 {
25     System.out.println("going to deposit...");
26     this.amount+=amount;
27     System.out.println("deposit completed... ");
28     notify();
29 }
30 }
31 public class inter_demo {
32
33     public static void main(String[] args) {
34         final Customer c=new Customer();
35         new Thread(){
36             public void run(){c.withdraw(15000);}
37         }.start();
38         new Thread(){
39             public void run(){c.deposit(10000);}
40         }.start();
41     }
42 }
43 }

```

Deadlock

In Java, deadlock is a type of multithreading. A deadlock can occur when one thread is waiting for an object lock acquired by another thread and the second thread is waiting for an object lock acquired by the first thread. The condition is known as deadlock because both threads are waiting for each other to release the lock.



```
eclipse-workspace - GL_PROJECT/src/com/gl/Dead_lock.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
string_demo... next_demo.java next_line_d... string_manip... TestMultiPr... no_sync_demo... sync_demo.java syn_block.java inter_demo.java Dead_lock.java
GL_PROJECT src com.gl Dead_lock
1 package com.gl;
2
3 class Pen{}
4 class Paper{}
5
6
7 public class Dead_lock {
8
9     public static void main(String[] args)
10    {
11        final Pen pn =new Pen();
12        final Paper pr =new Paper();
13
14        Thread t1 = new Thread() {
15            public void run()
16            {
17                synchronized(pn)
18                {
19                    System.out.println("Thread1 is holding Pen");
20                    try{
21                        Thread.sleep(1000);
22                    }
23                    catch(InterruptedException e){
24                        // do something
25                    }
26                    synchronized(pr)
27                    {
28                        System.out.println("Requesting for Paper");
29                    }
30                }
31            }
32        };
33
34        Thread t2 = new Thread() {
35            public void run()
36            {
37                synchronized(pr)
38                {
39                    System.out.println("Thread2 is holding Paper");
40                    try {
41                        Thread.sleep(1000);
42                    }
43                    catch(InterruptedException e){
44                        // do something
45                    }
46                    synchronized(pn)
47                    {
48                        System.out.println("requesting for Pen");
49                    }
50                }
51            }
52        };
53
54        t1.start();
55        t2.start();
56    }
57 }
```

```

50         }
51     };
52
53     t1.start();
54     t2.start();
55 }

```

Output:

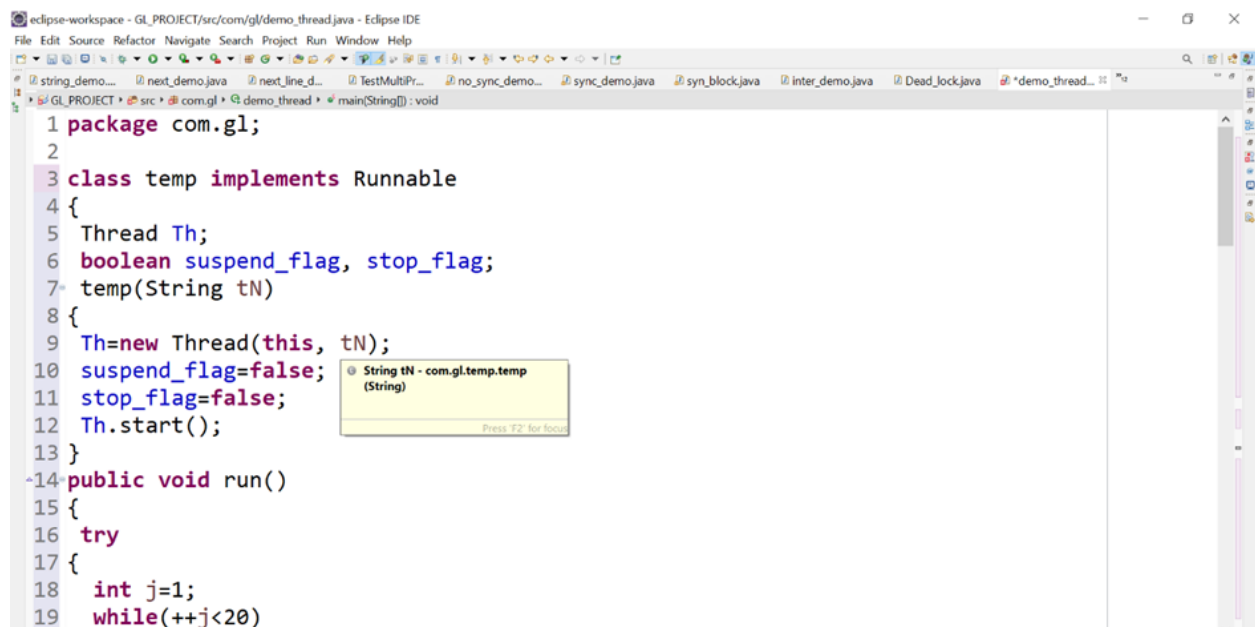
Thread2 is holding Paper
Thread1 is holding Pen

Suspending, Resuming and Stopping Threads

In a multithreading program, the functions of suspending, resuming, and stopping a thread are performed using Boolean-type flags. These flags are used to store the thread's current state.

1. If the suspend flag is set to true, run() will pause the current thread's execution.
2. If the resume flag is set to true, run() will resume the suspended thread's execution.
3. When the stop flag is set to true, the thread is terminated.

Examples:



The screenshot shows the Eclipse IDE with a Java project named 'GL_PROJECT'. The main editor displays the file 'demo_thread.java'. The code defines a 'temp' class that implements the 'Runnable' interface. It has a 'Thread' member 'Th' and two boolean flags: 'suspend_flag' and 'stop_flag'. The constructor 'temp(String tN)' initializes 'Th' with 'this' and 'tN', and sets both flags to false. The 'run()' method contains a 'try' block with a 'while(++j < 20)' loop. A tooltip for the variable 'tN' is visible, showing its type as 'String' and its value as 'com.gl.temp.temp (String)'.

```

1 package com.gl;
2
3 class temp implements Runnable
4 {
5     Thread Th;
6     boolean suspend_flag, stop_flag;
7     temp(String tN)
8     {
9         Th=new Thread(this, tN);
10        suspend_flag=false;
11        stop_flag=false;
12        Th.start();
13    }
14    public void run()
15    {
16        try
17        {
18            int j=1;
19            while(++j<20)

```

```

20 {
21     synchronized(this)
22     {
23         while(suspend_flag)
24         { wait();}
25         if(stop_flag)
26         { break;}
27     }
28 }
29 }
30 catch(InterruptedException IE)
31 {
32     System.out.println("Thread Interrupted");
33 }
34 }
35 synchronized void my_suspend()
36 {
37     suspend_flag=true;
38 }
39 synchronized void my_resume()
40 {
41     suspend_flag=false;
42     notify();
43 }
44 synchronized void my_stop()
45 {
46     suspend_flag=false;
47     stop_flag=true;
48     notify();
49 }
50 }
51
52 public class demo_thread {
53     public static void main(String args[])
54     {
55
56         Thread.sleep(2000);
57         t1.my_stop();
58         System.out.println("Thread demo is Stopped");
59     }
60     catch(InterruptedException IE)
61     {
62         System.out.println("Generated interrupted exception");
63     }
64 }
65 }
66 }

```

Output:

Thread demo is Created and Started
 Thread demo is Suspended
 Thread demo is Resumed
 Thread demo is Suspended
 Thread demo is Resumed
 Thread demo is Stopped